Detect bottlenecks

Identify problems with the cache usage

The use of the cache has a considerable impact on the overall system performance, even with small targets. This efficiency plays an important role, in particular in regard to real-time applications. Possibilities for optimisation should already be found in a very early phase of the software development lifecycle. The sooner the bottleneck is recognised, the easier and cheaper it can then also be removed. A procedure is suited for this task, which actually should be established in every development team: Static code analysis. With its use, it is possible to root out problems involving the cache usage and to remove them.

By Royd Lüdtke, Director of Static Analysis Tools at Verifysoft Technology GmbH

Two current trends pose considerable challenges to the developers in the automotive industry: On the one hand the requirements concerning safety and reliability of the systems are increasing. In this regard, the quality control along the entire development and life cycle of the systems plays an increasingly large role in order to ensure the functional and non-functional requirements as well as the safety of the systems on a high level. On the other hand, the need for real-time capable systems is growing. With autonomous vehicles and so-called “connected cars”, an ever-increasing amount of data must be recorded and processed. With this, the performance of the systems is becoming a critical factor. However, with embedded systems, the available hardware resources are manageable. In the framework of quality control, both the requirements set for reliability as well as the performance can already be addressed at a very early phase of the development.

Tools for static code analysis

The dynamic testing is the standard for the evaluation and verification of the systems. Additionally, the use of tools for the static code analysis has established itself as a method for the detection of errors or deviations from programming standards or guidelines. The operating principle is based on transferring the code, which is to be examined, into a model. The intermediate representation (IR), as it is also generated by compilers, is used for this. Based on this model, all possible data and control streams are run through and verified using defined checkers. The advantage of this approach is that no executable code needs to be present. Therefore, the static code analysis can already be utilised at a very early stage of the SDLC. Because the static analysis takes all states into account that the program can theoretically assume, potential errors can be identified with a very high hit ratio. For this reason, programming standards in safety-critical industry sectors such as DO-178 B/C in aviation, prescribe the use of this procedure; many other standards such as MISRA-C recommend it. The many errors that lie in the focus of the analysis, among others include the traditional gateways for malware:

- Buffer overrun/underrun
- Command injection
- Integer overflow of allocation size

- SQL injection
- Non-constant format string

The software development can also utilise this procedure away from the traditional programming errors, in particular with embedded systems. In the embedded development, the available hardware is often limited for cost reasons; the software must function in a high-performance manner with few resources. Here, the performance of a system is often unnecessarily slowed down as a result of minor issues. The cache is a typical bottleneck that only receives little attention: The cache is a particularly fast memory, which is used for the temporary storage of frequently required data. Memory implemented as SRAM (Static Random Access Memory) is up to 100-times faster than the DRAM (Dynamic Random Access Memory) usually used as the main memory, because with DRAM, capacitors must be loaded in a time-consuming manner. However, SRAM is considerably more expensive than DRAM and therefore is only used sparingly. Usually, the SRAM is limited to the relatively small cache integrated in the CPU. In addition to the memory technology, the size of the memory is also a decisive factor for the performance. Here, the following applies: The larger the memory capacity, the slower the access. For this reason, today often up to three cache levels are used.

Misfetches lead to losses in performance

The sequence is strictly hierarchical: The CPU initially searches for the required data in the first level cache (L1). When it is not available there, then this is referred to as a misfetch. Following an L1 misfetch, the search is performed in the second level cache (L2), subsequently in the third level cache (L3). When the L3 is also not able to provide the necessary data, the main memory must be accessed, which involves considerable losses in performance. The average access time increases with each cache level. Here, the cache is organized in such a manner that larger blocks of data can be addressed and read out at once. The access of these so-called cache lines – also referred to as cache blocks – avoids the bandwidth issues that are familiar from the system bus. The size of the cache lines depends on the processor type, values between 64 and 128 bytes are customary.

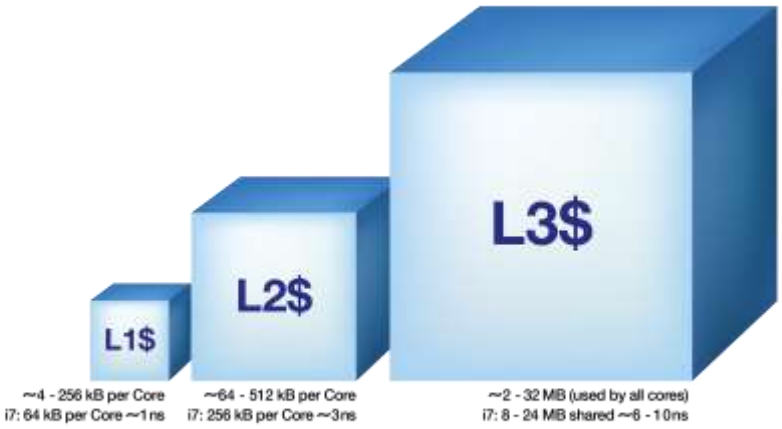

Image caption: With the size of the cache, the average access time also increases.

Therefore, when more of the frequently required data can be provided on a lower cache level, the likelihood of misfetches will be reduced. As a result, an optimally utilised cache is beneficial for the overall performance of the system. The developer can actively influence how the cache is occupied. A typical code example that can lead to performance problems can appear as follows:

```
struct Test_Struct
{
      char a;
      int b;
      char c;
      float d;
      char e;
} ;
struct Test_Struct *precord;
…
for (i = 0; i < 1024 * 1024 * 10; i++)
{
      precord[i].a++;
      precord[i].b++;
      precord[i].c++;
      precord[i].d++;
      precord[i].e++;
}
```

The example features a relatively large array of structures in five variables, in which different data types can occur in a mixed manner. When these variables must be accessed often, their representation in the cache and therefore in the cache line would appear as follows:

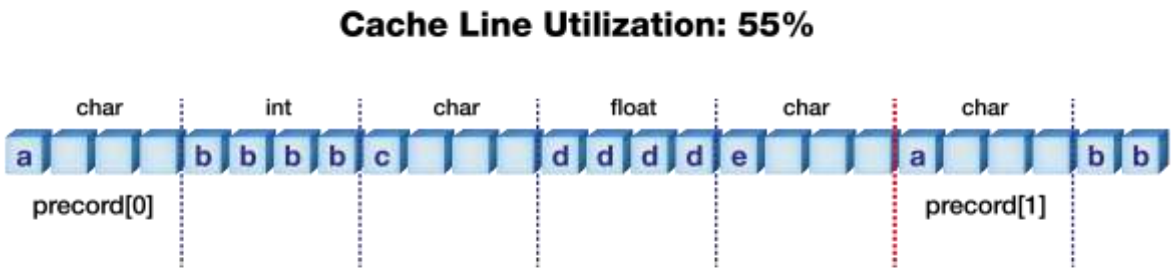

Image caption: The system performance is not optimal, as a considerable part of the cache line remains unused and misfetches are to be expected.

Due to the fact that on 32-bit systems both the cache line as well as the main memory are organized in words with a word size of four bytes (alignment), large sections remain unused: The first variable “a” of the type `char` occupies one byte and is stored in the first word. The next variable “b” of the type `int` requires four bytes; it must be stored in the second word. The other variables behave in an

analogous manner. The result: In this example, only 55 percent of the cache line is used. As a result, the required amount of cache access instances increases by 45 percent. With this, the likelihood also increases that the required data is not stored in this cache and therefore must be loaded from slower memory such as the main memory.

Remedy by resorting

This performance disadvantage can be removed by the resorting of the variables in the code: When five variables are arranged according to their type – and therefore their storage requirements – in the cache, then the cache can be used in a significantly more effective manner.

```
   struct Test_Struct
{
      char a;
      char c;
      char e;
      int b;
      float d;
} ;
```

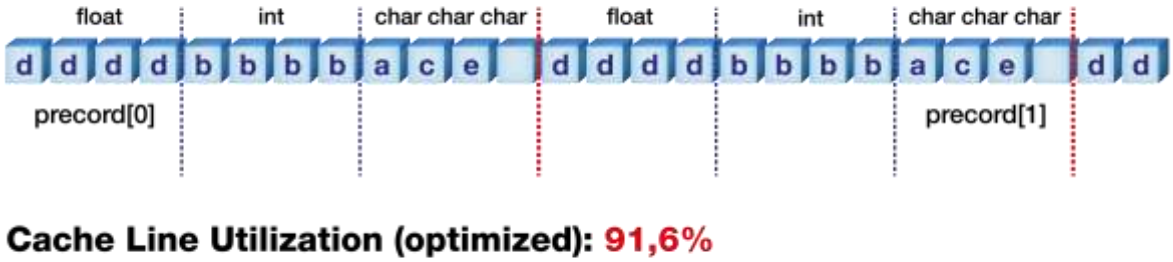

Image caption: When the variables in the code are sorted according to their storage requirements, the cache line is used in a virtually optimal manner.

Initially, in our small example this does not look like a sensational gain. However, it should be taken into consideration that each change of individual bits of the cache is required in order to write the entire cache line back to the main memory via the system bus. This of course also applies to empty bytes. As a result of the optimisation, a significantly smaller portion of the limited bus bandwidth is claimed by the application. Here, a positive side effect occurs: In addition to the improved performance of the correctly used cache, a reduced memory requirement of the application in the main memory occurs, the so-called memory footprint.

However, not only the manner in which the variables are sorted in the code can lead to noticeable performance weaknesses. Inefficient data access also leads to the application being slower than necessary.

```
struct Test_Struct
{
      int a;
      int b;
      int c;
      int d;
      int e;
} ;
struct Test_Struct *precord;
…
for (i = 0; i < 1024 * 1024 * 50; i++)
{
      precord[i].b++;
}
```

The result is again a very unfavourable use of the cache line, as shown in the distribution of the structure variables:

Image caption: Although the cache in principle is filled optimally, the majority of the data contained in it remains unused.

Although no memory space is wasted, the usage is not optimal, because only a single element is accessed within each structure instance. By dividing the array into two arrays or by creating a second, temporary array, a significant acceleration can be achieved here.

```
struct Test_Struct
{
      int a;
      int c;
      int d;
      int e;
} ;
```

```
struct Test_Struct *precord;
int *b;
…
for (i = 0; i < 1024 * 1024 * 50; i++)
{
      b[i]++;
}
```

The result of this optimisation is the complete use of the cache line, which now solely provides the required data. In practice, this form of optimisation, depending on the size of the array, has a significant effect on the performance of an application.

Tools for the static analysis can generally by expanded with individual checkers, which are relatively easy to implement. To find performance problems such as unused memory space in the cache line, the checker must recognise whether the structure and member variables are arranged as ideally as possible. This also includes the recognition whether a new arrangement of the variables will only have a positive effect on the memory footprint of the application or if additionally significant performance improvements are also to be expected. This question among others depends on whether this data must be accessed often. For example, this is the case in loops.

The sooner, the better

The last-mentioned example, the structure that is merely partially used, requires the splitting up into arrays to achieve the optimization. Generally, this approach entails comprehensive changes in the code, therefore it can only be implemented in an economically viable manner in a relatively early phase of the development. With the use of a tool for static analysis, the problem can be identified early on and as a result solved with acceptable costs. Possible performance improvements of a significant extent justify this approach. Additionally, the static analysis during the performance optimisation is not limited to the caching. It is also conceivable to identify loops with this, which can be merged or to point out functions that can be condensed into a single function. Numerous checks are possible here. However, the following must be clear: The performance optimisation with the use of the static analysis cannot and will not replace systematic, methodical software performance engineering. However, the approach is well suited in already eliminating potential bottlenecks during the development. The following generally applies: The sooner a problem is identified, the less effort and costs are involved in its correction. Only performing the verification of non-functional requirements as late as the testing can often have devastating consequences. Extensive changes of the software will then hardly be useful by that point. Each small and large weak point in the code that has been corrected by that stage is therefore a gain – for the development budget, the code quality and of course for the time to market.